Attorney Docket No. SUN1P809/P5500

PATENT APPLICATION

IMPROVED FRAMEWORKS FOR INVOKING METHODS IN VIRTUAL MACHINES

Inventors: 1. Stepan Sokolov
34832 Dorado Common
Fremont, CA 94555
Citizenship: Ukraine

Assignee: Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303

BEYER WEAVER & THOMAS, LLP
P.O. Box 778
Berkeley, CA 94704-0778
Telephone (650) 961-8300

IMPROVED FRAMEWORKS FOR INVOKING METHODS IN VIRTUAL MACHINES

CROSS-REFERENCE TO RELATED APPLICATIONS

This application is related to U.S. Patent Application No. _____ (Att.Dkt.No. SUN1P810/P5510), entitled "IMPROVED METHODS AND APPARATUS FOR NUMERIC CONSTANT VALUE INLINING IN VIRTUAL MACHINES", filed concurrently herewith, and hereby incorporated herein by reference.

This application is related to U.S. Patent Application No. _____ (Att.Dkt.No. SUN1P814/P5417), entitled "IMPROVED FRAMEWORKS FOR LOADING AND EXECUTION OF OBJECT-BASED PROGRAMS", filed concurrently herewith, and hereby incorporated herein by reference.

BACKGROUND OF THE INVENTION

The present invention relates generally to frameworks for invoking methods in virtual computing machines. More specifically, the invention relates to frameworks for representing class files that facilitate the method invocation process within a virtual machine and to processes for creating such representations of a class file.

Recently, the Java™ programming environment has become quite popular. The Java™ programming language is an object-based high level programming language that is designed to be portable enough to be executed on a wide range of computers ranging from small devices (e.g., pagers, cell phones and smart cards) up to supercomputers. Computer programs written in the Java programming language (and other languages) may be compiled into Java virtual machine instructions (typically referred to as Java bytecodes) that are suitable for execution by a Java virtual machine implementation.

The Java virtual machine is commonly implemented in software by means of an interpreter for the Java virtual machine instruction set, but in general may be software, hardware, or both. A particular Java virtual machine implementation and corresponding support libraries, together constitute a Java™ runtime environment.

Computer programs in the Java programming language are arranged in one or more classes or interfaces (referred to herein jointly as classes or class files). Such programs are generally platform, i.e., hardware and operating system, independent. As such, these computer programs may be executed, unmodified, on any computer that is able to run an implementation of the Java™ runtime environment. A class written in the Java programming language is compiled to a particular binary format called the "class file format" that includes Java virtual machine instructions for the methods of a single class. In addition to the Java virtual machine instructions for the methods of a class, the class file format includes a significant amount of ancillary information that is associated with the class. The class file format (as well as the general operation of the Java virtual machine) is described in some detail in The Java Virtual Machine Specification by Tim Lindholm and Frank Yellin (ISBN 0-201-31006-6), which is incorporated herein by reference.

As described in The Java Virtual Machine Specification, one of the structures of a standard class file is known as the "Constant Pool." The Constant Pool is a data structure that has several uses. One of the uses of the Constant Pool that is relevant to the present invention is that the Constant Pool contains the information that is needed to resolve each method that can be invoked by any of the methods within the class. Fig. 1 (which may be familiar to those skilled in the art) is a representation of a Constant Pool section of a class file that contains the information necessary to uniquely identify and locate a particular invoked method.

Generally, when a class file is loaded into the virtual machine, the virtual machines essentially makes a copy of the class file for its internal use. The virtual machine's internal copy is sometimes referred to as an "internal class representation." In conventional virtual machines, the internal class representation is typically almost an exact copy of the class file and it replicates the entire Constant Pool. This is true regardless of whether multiple classes loaded into the virtual machine reference the same method and thus replicate some (or much) of the same Constant Pool information. Such replication may, of course, result in an inefficient use of memory resources. In some circumstances (particularly in embedded systems which have limited memory resources) this inefficient use of memory resources can be a significant disadvantage.

Additionally, conventional virtual machine interpreters decode and execute the virtual machine instructions (Java bytecodes) one instruction at a time during execution, e.g., "at runtime." To invoke a method referenced by a Java bytecode, the virtual machine must make access the Constant Pool simply to identify the information necessary to locate and access the method to be invoked. Again, this is inefficient. Such inefficiencies tend to slow the performance of the virtual machine. Accordingly, improved frameworks for invoking methods in virtual machines such as Java virtual machines would be very useful.

SUMMARY OF THE INVENTION

To achieve the foregoing and other objects of the invention, improved frameworks for implementing class files that are particularly useful in facilitating the method invocation process within a virtual machine will be described. In one aspect of the invention, a class file structure is described which associates one or more "reference cells" with the class file. Generally, each method within the class file (or an internal representation of the class file) may have an associated reference cell. The reference cells typically include sufficient information to facilitate the invocation of the corresponding method.

In one embodiment, each reference cell includes a class pointer field, a method name field and a signature field. The class pointer field can contains a reference to an internal class representation. The method name field contains or references the name of the associated method. The signature field contains or references a signature associated with the method. The reference cell may include other information as well. By way of example, some implementations may further include an information field and a link field. The information field is arranged to hold or reference information generated at runtime by the virtual machine. The link field contains information suitable for directly or indirectly linking the reference cell to the internal class representation.

In various preferred embodiments, the class file takes the form of an internal class representation suitable for direct use by a virtual machine. By way of example, the internal class representation may represent a Java class. In such embodiments, the internal class representation preferably does not include a Constant Pool.

In some preferred embodiments, the signature takes the form of an internal representation of a signature that is directly useable by the virtual machine at runtime. That is, the signature is stored in the reference cell in a form that does not require construction of the signature by the virtual machine.

In another aspect of the invention, a process for loading class files into a virtual machine based computing system is described. Each method invocation within a class file is translated into an internal method invocation that references a reference cell associated with the internal class representation that contains the method. The use of such reference cells has the potential in many circumstances to improve the

performance of the virtual machine as well as to potentially reduce the memory requirements of the internal class representations.

In yet another aspect of the invention, a process for creating internal representations of a class file is described. In this aspect, each method invocation in a class file is reviewed to determine whether a reference cell currently exists for its associated method. When it is determined that a reference cell does not currently exist for a method associated with a selected method invocation, a new reference cell is created for the selected method. The newly created reference cell is then associated with the internal class representation that contains the method (which may or may not be different than the internal class representation that contains the method invocation). In various preferred embodiments, the internal class representation represents a Java class and does not include a Constant Pool.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention will be readily understood by the following detailed description in conjunction with the accompanying drawings, wherein like reference numerals designate like structural elements, and in which:

Fig. 1 is a representation of a Constant Pool section of a class file that contains the information necessary to uniquely identify and locate a particular invoked method.

Fig. 2 is a block diagram of an internal class representation in accordance with one embodiment of the invention.

Fig. 3 is a diagrammatic representation of a reference cell in accordance with one embodiment of the present invention.

Fig. 4 illustrates an exemplary loading method for loading a class file in accordance with one embodiment of the invention.

Fig. 5 the internal class representation illustrates a method for processing a method invocation in accordance with one embodiment of the invention.

Fig. 5B illustrates a conventional Java Method Invocation.

Fig. 5C illustrates a Java Method Invocation in accordance with one embodiment of the invention

Fig. 6 illustrates a method for creating a reference cell in accordance with one embodiment of the present invention.

DETAILED DESCRIPTION OF THE INVENTION

As described in the background section, the Java programming environment has enjoyed widespread success. Therefore, there are continuing efforts to extend the breadth of Java compatible devices and to improve the performance of such devices. One of the most significant factors influencing the performance of Java based programs on a particular platform is the performance of the underlying virtual machine. Accordingly, there have been extensive efforts by a number of entities to provide improved performance Java compliant virtual machines. In order to be Java compliant, a virtual machine must be capable of working with Java classes, which have a defined class file format. Although it is important that any Java virtual machine be capable of handling Java classes, the Java virtual machine specification does not dictate how such classes are represented internally within a particular Java virtual machine implementation.

The Java class file format utilizes the constant pool construct to store the information necessary to support method invocation. However, as suggested above, traversing the constant pool at runtime to obtain the information necessary to invoke a method is not particularly efficient. One aspect of the present invention seeks to provide a mechanism that will generally improve the runtime performance of virtual machines by eliminating the need to always traverse a constant pool at runtime to invoke particular methods. Conventionally, a significant amount of work has to performed at runtime in order to invoke a method. In effect, the described system contemplates doing some extra work during the loading of a class into a virtual machine by obtaining the method invocation information from the constant pool during loading and representing that information in a form that is more efficient at runtime.

In other aspects of the invention, specific data structures (i.e., internal representations of a class file) that are suitable for use within a virtual machine and methods for creating such internal representations of a class file are described. In one embodiment, the improved internal class representation includes a reference portion associated with the various methods that are contained within the class. The reference

portion includes a plurality of reference cells, wherein each of the reference cells corresponds to a unique method associated with the class. The reference cells are used to store information necessary for a virtual machine to invoke their corresponding methods at run time. With this arrangement, methods can be invoked using reference cells without requiring constant pools to be traversed at runtime. Further, if the invention is implemented together with other improvements as described in concurrently filed, co-pending U.S. Patent Application No. _____ (Att.Dkt.No. SUN1P814/P5417), entitled "IMPROVED FRAMEWORKS FOR LOADING AND EXECUTION OF OBJECT-BASED PROGRAMS", the constant pool may not even need to be copied into the virtual machine's internal representation of the class file. Thus, the use of reference cells in conjunction with the improved techniques provided in accordance with the invention has the potential in many circumstances to improve the performance of the virtual machine as well as potentially reduce the memory requirements of the internal class representations.

Embodiments of the invention are discussed below with reference to Figs. 2 - 6. However, those skilled in the art will readily appreciate that the detailed description given herein with respect to these figures is for explanatory purposes as the invention extends beyond these limited embodiments.

Fig. 2 is a block diagram of an internal class representation 200 in accordance with one embodiment of the invention. The internal class representation 200 can be, for example, implemented as a data structure embodied in a computer readable medium that is suitable for use by a virtual machine. As shown in Fig. 2, the internal class representation 200 includes a method information portion 202. The method information portion 202 is arranged to contain or reference information relating to one or more methods. The methods can be, for example, Java implemented methods. As will be appreciated, the method information portion 202 can be implemented in various ways, for example, as a table similar to a table of method information implemented in a standard Java class file.

In addition, the internal class representation 200 includes a method reference portion 204 associated with the methods contained within the internal class representation 200. The reference portion 204 is arranged to include any reference cell associated with the class. The method reference portion 204 can be implemented

in a wide variety of different ways depending on the needs of a particular system. By way of example, in the described embodiment, the reference cells are linked together using a link-list construct. Each reference cell can include information that is useful in invoking a method.

It should be noted that reference cells can be created selectively (e.g., when it is certain and/or when it is likely that a method is to be invoked). Accordingly, in accordance with one embodiment of the invention, if a method does not appear to be invoked, reference cells corresponding to that method need not be created. As a result, processing time and memory space may further be improved.

Fig. 3 is a diagrammatic representation of a reference cell 300 in accordance with one embodiment of the present invention. In the illustrated embodiment, the reference cell 300 includes a class pointer field 302, a method name field 304, a signature field 306, an information field 308, and a link field 310. Typically, a value stored in the class pointer field could reference (i.e. point to) an internal representation of class. The class may have one or more methods associated with it. A method can have a corresponding reference, for example, the reference cell 300. The method name field 304 is arranged to identify the name of a method associated with the class. This is typically done be storing the actual method name in the method name field 304. However again, this can also be accomplished by storing a pointer or index to the method name.

The signature field 306 is arranged to contain or reference a signature associated with the method corresponding the reference cell. The nature of the signature is well known to those familiar with method invocation in Java virtual machines. Typically, in most conventional Java virtual machines, the signature is constructed into a form usable by the virtual machine using a series of calls to the constant pool at runtime when the method is invoked. Although this works well, it is relatively slow. One advantage to including the signature in the reference cell is that a signature suitable for direct use by the virtual machine can be constructed during loading and either stored directly in the reference cell or stored in a location that is referenced by the reference cell. In the described embodiment, the reference cell contains a pointer or index to the signature rather than actually storing the signature simply because signatures can be relatively large and their relative sizes may vary

significantly for different classes. In addition, references to signatures can be used across reference cells. Thus, referencing the signature tends to be a more efficient use of memory.

The information field 308 is arranged for containing or referencing information generated at runtime by the virtual machine. As will be appreciated by those skilled in the virtual machine art, there is often information associated with a method that the virtual machine desires to store at runtime. The information field 308 simply provides a place to either store such information, or to identify the location where such information can be or is stored.

In the illustrated embodiment, the reference cells are associated with their corresponding class using a link list construct. Thus, the link field 310 simply contains information suitable for directly or indirectly linking the reference cell 300 to the internal class representation and/or other associated reference cells.

As briefly discussed above, method invocations in standard Java class files take the form of a method invocation instruction followed by a data parameter which takes the form of an index into the constant pool. To execute a method invocation at runtime, the virtual machine (e.g. the virtual machine interpreter), must perform a number of steps which require multiple inquiries to the constant pool. With the framework described herein, much of this work can effectively be transferred to class loading by creating the described reference cell architecture during class loading.

Fig. 4 illustrates an exemplary loading method 400 for loading a class file in accordance with one embodiment of the invention. For the sake of illustration, in the described embodiment, the loading method 400 is used in a Java runtime environment to load a Java class file. Typically, this would be accomplished by a Java class loader. Initially, at operation 402, a determination is made as to whether an internal class representation shell exists for the class being loaded (e.g., a class L). If it is determined at operation 402 that an internal class representation shell does not exist for the class L, an internal class representation shell is created for class L in operation 404. However, if it is determined at operation 402 that an internal class representation shell exists for the class L, the loading method 400 skips operation 404 and proceeds directly to operation 406 where the internal class representation shell for class L is populated. In the disclosed embodiment, the class loader populates the internal class

representation using any appropriate techniques. However, the method invocations are handled differently then method invocations are typically handled during loading in that the described reference cell framework is created. Populating the internal class representation performed at operation 406 can entail identifying and processing Method invocations (operation 408) which is illustrated in greater detail in Fig. 5. After the internal class representation has been populated, any other class loading related processing with respect to Class L is undertaken in operation 410 and the loading is completed. In one preferred embodiment, class loading may be performed, as described in concurrently filed, co pending U.S. Patent Application No. _____ (Att.Dkt.No. SUN1P814/P5417), entitled "IMPROVED FRAMEWORKS FOR LOADING AND EXECUTION OF OBJECT-BASED PROGRAMS".

Referring next to Fig. 5 the internal class representation illustrates a method 408 for processing a method invocation in accordance with one embodiment of the invention. The method 408 represents operations that can be performed at operation 408 of Fig. 4. It should be noted that the method 408 can be utilized by a virtual machine to process method invocations in a programming environment (e.g., Java programming environment). To illustrate, the processing method 408 will be described below with respect to a Java programming environment.

Initially, at operation 502, the next Java bytecode is read. Next, at operation 504, a determination is made as to whether the next Java bytecode is a method invocation, for example, a method invocation M for invoking a method M. As illustrated in Fig. 5B, the method invocation is typically a Java "*invoke*$_x$M (*cp-index*)". "(*cp-index*)" is an index to a constant pool in a Java class file associated with a class M. Accordingly, "(*cp-index*)" is used to obtain information relating to the method M. This information includes class name, method name, and signature for a method M. If it is determined at operation 504 that the next Java bytecode is not a method invocation, the method 408 proceeds to operation 506 where the bytecode is processed. The method 408 then proceeds to operation 520 where a determination is made as to whether there is a Java bytecode to be read. If it is determined at operation 520 that there is at least one bytecode to read, the method 408 proceeds back to operation 502 where the next Java bytecode is read. However, if it is determined at operation 520 that there are no more bytecodes to read, the method 408 ends.

On the other hand, if it is determined at operation 504 that the next Java bytecode is a method invocation, the method 408 proceeds to operation 508 where the constant pool information from the Java class file for the Java class M is read. As noted above, the index (*cp-index*) of Java "*invoke*$_X$_M (*cp-index*)" can be used to obtain this information from the constant pool. It should be noted that the Java class M is a class associated with the method M. It should also be noted that class M is a class associated with a class that is to be loaded such as class L of the loading method 400 of Fig. 4.

At operation 510 a determination is made as to whether an internal class representation shell exists for the identified class M associated with the method M. If it is determined at operation 510 that an internal class representation shell does not exist for the identified class M, the method 408 proceeds to operation 512 where a class representation shell for the identified class M can be created. Following operation 512 or if it is determined at operation 510 that an internal class representation shell does exist for the identified class M, the method 408 proceeds to operation 514 where a determination is made as to whether a reference cell M exists for the identified method M. The reference cell M can be implemented, for example, in accordance with the reference cell 300 of Fig. 3. If it is determined at operation 514 that a reference cell M does not exist for the identified method M, the method 408 proceeds to operation 516 where a reference cell M for the identified method M is created. However, If it is determined at operation 514 that a reference cell M exists for the identified method M, the method 408 skips operation 516 and proceeds directly to operation 518 where the method invocation is reconstructed. As illustrated in Fig. 5C, a "Java *invoke*$_X$_M (*cp-index*)" instruction can be reconstructed to a *invoke*$_{AVM}$_M (*reference M*) instruction wherein the "*reference* M" is a reference (or pointer) to a reference cell associated with the method M. At operation 620 a determination is made as to whether there are more Java bytecodes to be read. If it is determined at operation 520 that there is a Java bytecode to be read, the method 408 proceeds back to operation 502 where the next Java bytecode is read. However, if there are no more bytecodes, the method 408 ends.

Fig. 6 illustrates a method 600 for creating a reference cell in accordance with one embodiment of the present invention. The method illustrates in greater detail

operations that can be performed at operation 516 of Fig. 5. Initially, at operation 602, a shell for a reference cell M is created. Next, at operation 604, a class pointer field of a reference cell is filled with (or set to) the appropriate internal class representation. For example, as shown in Fig. 3, the reference cell M can be the reference cell 300 having a class pointer field 302.

At operation 606, the method name field of the reference cell is filled with (or set to) the appropriate method name for a method M associated with the class M. For example, referring back to Fig. 3, the method name field 304 can be filled with (or set to) the appropriate value, namely, the method name for the method M. As will be appreciated, if necessary, the appropriate method name can be stored.

Next, at operation 608, a determination is made as to whether an internal representation exists for the signature of the method associated with the cell reference. If it is determined at operation 608 that an internal representation does not exist for the signature of the method associated with the cell reference, the method 600 proceeds to operation 610 where the original representation of the signature is parsed. The original representation of the signature is typically found in a Java class file containing an invocation method M. At operation 612, an internal representation for the signature of the method associated with the cell reference is created. As will be appreciated, if necessary, the internal representation for the signature can be stored.

Following operation 612 or if it is determined at operation 608 that an internal representation exists for the signature of the method associated with the cell reference, the method 600 proceeds to operation 614 where the signature field of the reference cell M is filled with (or set to) the appropriate internal representation of the signature of method M. For example, referring back to Fig. 3, the signature field 306 can be filled with (or set to) the appropriate value, namely, the internal representation of the signature for the method M. As will be appreciated, if necessary, the internal representation of the signature of method M can be stored.

Following operation 616, if necessary, a link field of the reference cell M is updated. For example, referring back to Fig. 3, the link filed 308 of reference cell 300 of Fig. 3 is updated. The link filed of the reference cell M can, for example, be updated to point to another cell reference in a linked list of cell references. The method 600 end following operation 616.

It should be noted that it is not necessary for the internal class representations implemented in accordance with the invention to have constant pool portions. Furthermore, the information relating to a method can be generated at load time and provided in a reference cell to allow invocation of the method. In comparison to conventional internal class representation there is no a need to copy constant pools into a virtual machine. Furthermore, there is no need to access constant pools and perform various operations at run time to obtain the information necessary to resolve methods.

The many features and advantages of the present invention are apparent from the written description, and thus, it is intended by the appended claims to cover all such features and advantages of the invention. Further, since numerous modifications and changes will readily occur to those skilled in the art, it is not desired to limit the invention to the exact construction and operation as illustrated and described. Hence, all suitable modifications and equivalents may be resorted to as falling within the scope of the invention.

What is claimed is: